

FAQ

Q How does a markup language work with traditional code?

A A markup language describes the look of an application; the code controls what it does. Separating the two makes sense from a design point of view, as a designer can use markup language, which is easier to understand, to create the application's look and feel, while a programmer handles how it behaves. This also makes it easier to give an application a new look without rewriting it.

The internet is similar. Designers use the markup language HTML to design the site, while technologies such as JavaScript, PHP and .NET provide the internal workings.

Microsoft extended the idea with .NET and 'code behind'. An ASP .NET webpage has an HTML file to define its look and a C# or VB file to say what happens when a button is pressed.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Extending Firefox

Have you ever wished that Firefox could do more? Mike James shows you how to write an extension so that you can get your browser to do what you want



Firefox's strength isn't simply that it's a secure, stable web browser; it has also been designed so that it's easy to create add-ons – or extensions – to extend its capabilities. The add-ons website (<http://addons.mozilla.org>) has thousands of examples for download, including pop-up blockers, add-ons that enable it to render certain pages using Internet Explorer's engine and an updated tab manager.

Don't worry if you can't find the add-on you want, though, as it's fairly easy to create your own. Firefox uses a subset of the Gecko layout engine, called Chrome, which gives you a framework for building your own add-ons using XML and JavaScript. It's platform-independent, so anything you create will work the same way under Windows, Linux and Mac OS.

Even better, the free Mozilla Thunderbird mail client uses the same engine, so it's just as easy to create add-ons for that application. This month we'll concentrate on creating a simple Firefox extension, but the principles we teach will help you with Thunderbird, too.

THE GREAT IDEA

A Firefox extension has two major components: XUL, pronounced 'zool', which defines the user interface; and XPCOM, a JavaScript API that provides access to the internal components. Most guides to building extensions focus almost exclusively on XUL, perhaps because this is the more impressive of the two components. To get anything done, however, you need to know about XPCOM and the library of components to which it provides access. Here we will examine how both work. If you know how ASP, .NET or WPF work, it might help to compare XUL with HTML or XAML and XPCOM with the .NET Framework classes.

XUL is an XML-based markup language that extends HTML – or, more accurately, XHTML. This has a huge and initially unexpected advantage in that it blurs the distinction between the data that the application is displaying and the application's user interface. In Firefox, for example, you can access the

HTML page being displayed from script using the Document Object Model (DOM). However, you can also access the entire user interface – status bar, toolbar, menus and so on – in exactly the same way. This means that if you know how to use the DOM, you have access not only to the webpage on display but to the entire application's user interface.

This seems even more surprising, but is no less true, for Thunderbird, which apparently has nothing to do with browsers. In Thunderbird's case the DOM includes all the emails on display, which are simply HTML documents, and the entire user interface. Again, once you learn how to use the DOM in Firefox the same concepts and skills apply to Thunderbird.

PACKAGES

When creating an extension, the first hurdle is to implement the way the code is bundled into a Zip file, also called a Bundle, with the extension .xpi (pronounced 'zippy'). This uses a fixed folder structure so that Firefox can find the information it needs to install the extension automatically. This is a good idea, but the structure is complicated and getting it right can

Firefox/Thunderbird Extension Wizard

The Extension Wizard generates an extension skeleton to meet your needs. It is intended to save the time you would normally spend copying and pasting from your previous extensions or sample extensions.

* = Required

Your Name: *

Extension Name: *

Extension Short Name: *

Extension ID: *

Version: *

Description:

Extension Homepage:

Update URL:

Contributors:

Additional Features:

▲ The online Firefox/Mozilla Extension Wizard is the easy way to generate a .xpi file for a project

be difficult. The simplest solution is either to download an existing extension and unzip it or to use the online Extension Wizard (<http://ted.mielczarek.org/code/mozilla/extensionwiz>).

However you create the .xpi file, you will need to provide some basic information about your application. If you use the Wizard, you simply fill in a form and it generates a Zip file that you can then download to a project directory. A Zip file for the 'Hello World' project that we'll write in a moment is included on the cover disc as helloworld.zip. To work with the project, simply unzip this file to create all the folders and template files you need.

If you open the project's main folder – helloworld in this case – you'll find a number of files and subfolders that you can investigate as you learn more about building extensions. At this stage the only folder you need to worry about is Content and the two files it contains: FirefoxOverlay.xul and overlay.js. The first is the XUL user interface definition, and the second is the JavaScript code for your extension.

For a simple example of an extension, let's add a new menu item to the Tools menu that simply pops up an alert window and says 'hello world'. The Wizard can create some XUL and code that does something similar but not in the simplest way, so let's create our own layout and code. First open the file FirefoxOverlay.js in Notepad and change it to read as follows:

```
<?xml version="1.0"
encoding="UTF-8"?>
<!DOCTYPE overlay SYSTEM
"chrome://helloworld/locale/
helloworld.dtd">
<overlay id="helloworld-overlay"
xmlns="http://www.mozilla.org/
keymaster/gatekeeper/there.is.
only.xul">
```

These three lines always begin an XUL file and simply establish its type and name. The third line specifies the id of your overlay. There are additional 'header' lines that you can include in the XUL file, but these are the most important. Next we can specify a file that contains JavaScript to be run when the overlay is installed. In our case, this is just the overlay.js file that the Wizard created:

```
<script src="overlay.js"/>
```

Now we can add lines that look a lot like HTML to define menu items, buttons, text boxes and so on. These will be added to the browser's user interface, but first we have to say exactly where we want our new controls to appear. As already mentioned, the entire browser user interface, including any HTML pages it might be displaying, forms a single huge DOM tree, and all we have to do is specify the id of the node to which we want to attach our controls.

The only problem is discovering the id of the existing control. You might think that the answer would be to look it up in the documentation, but in practice it is usually easier to use the DOM Inspector, which is explained later. The Tools menu has the id 'menu_ToolsPopup' and this is where our overlay starts:

```
<menupopup id="menu_ToolsPopup">
<menuItem
id="helloworld-hello"
```

```
label="Hello world"
oncommand=
"hello world.
onMenuItemCommand();" />
</menupopup>
</overlay>
```

The new menu item has the id 'helloworld-hello'. It's important that it uses a unique id to avoid conflicts within the DOM. You can add additional attributes to determine the behaviour and look of the new menu item. Setting the oncommand attribute determines the name of the function called when the user clicks the menu option. In this case it's the onMenuItemCommand method of the helloworld object, both of which are defined in the companion JavaScript file.

That's all there is to creating the XUL file for the overlay. To add or modify other elements of the user interface, all you need to do is look up the XUL element at www.xulplanet.com.

The next step is to write the JavaScript we need. Open the overlay.js file in Notepad and change it to read as follows:

```
var helloworld = {
  onLoad: function() {
    // initialisation code
    this.initialised = true;
  },
  onMenuItemCommand: function() {
    alert("Hello World");
  },
};
window.addEventListener("load",
function(e) { helloworld.onLoad(e);
},
false);
```

As long as you know a little JavaScript this should make sense. The first part creates a JavaScript object with two methods, onLoad and onMenuItemCommand. The onLoad function simply sets a local variable initialised to true so the other methods can check it has been initialised and that everything is loaded and ready to go. The onMenuItemCommand simply pops up an alert box with the message in it. This method has already been linked to the menu item event in the XUL file. The onLoad method, however, still needs to be linked to a suitable event. This is the job of the final line, which adds the onLoad method to the list of methods that are called when the browser's 'load' event fires.

TRYING IT OUT

All that remains is to try it all out. When you have finished writing your extension you distribute it for installation as an XPI file. To make this, you Zip all the directories that the wizard created into a single HelloWorld.zip file – like the one you downloaded – and then change the file's extension to .xpi. Firefox can then load this, either locally or from a webpage, and it automatically installs the extension using all the information in the XPI file.

This works well, but when you're testing an extension it's rather time-consuming to have

to create a Zip file and rename it every time you make any changes. To allow for development work, Firefox will load an unzipped extension directly from the development folders. All you have to do is place a text file containing the path to the development directory – the Firefox extensions directory. The first problem is finding the extensions directory. It's generally in:

C:\Documents and Settings\User name\
Application Data\Mozilla\Firefox\Profiles\some
unique letters.default\extensions

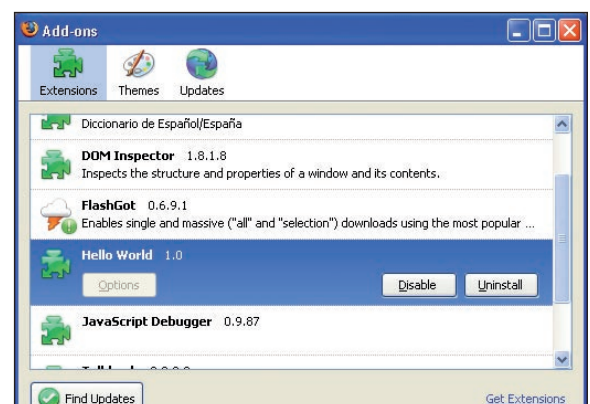
You must give the file that you store in the extensions directory the unique name you supplied to identify the extension, in this case helloworld@mike.james. If you supplied a different unique name for your extension, use that instead. Open Notepad, type in the path to the helloworld directory – in other words, the directory that contains the files chrome.manifest and install.rdf, among others – and save it as helloworld@mike.james. (To save a file that doesn't end in .txt using Notepad, surround the entire name in double quotes.)

Once you have saved the file in the appropriate directory, restart Firefox. If your extension has loaded properly, you should see it in the Add-ons manager, which you can open from the menu Tools, Add-ons. If you can't see it in the list, either you haven't restarted Firefox (try shutting all Firefox windows and restarting the application), or the file that specifies its path is in the wrong directory, or the path it specifies is incorrect. If you see your extension listed but an error message appears, something is wrong with the installation files. An extension will generally load even if there are errors in the XUL or JavaScript file. However, XUL or JavaScript files that have syntax errors will not load, even if your extension looks as if it has loaded without problems. This can result in an extension that looks as if it should do something, but nothing much happens when you try to use it. You can use the script debugger to see if the XUL and JavaScript files have loaded.

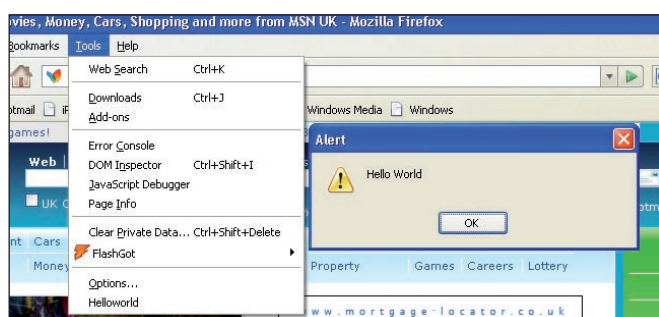
As long as the extension has loaded, you can try it out. If you click on the Tools menu, you should see the new "Helloworld" menu item. Clicking on this should produce your message in an Alert dialog box.

DOM INSPECTOR

Now that you have the beginnings of an extension, the rest is just more of the same: you edit the XUL file to add controls to the user interface, and you edit the .JS file to add behaviours. This is where the documentation



▲ The Hello World extension will appear in Firefox's Add-ons window



▲ The extension in action



▲ Existing controls, such as the status bar, can be controlled or added to

leaves you to get on with the job. There is, however, more to know if the job is going to be easy. First, you need to use the DOM Inspector. Installing this is an option when you first install Firefox. If you missed it, reinstall Firefox, select custom install and choose to install the DOM Inspector. You can also use the DOM Inspector with Thunderbird, but the easiest way to do this is to go to the Extensions download page, search for DOM Inspector, download the relevant extension and install it.

Once you have installed the DOM Inspector you can start it running using Tools, DOM Inspector. You can use it to discover the type and ID of any user interface control simply by clicking the icon in the top-left of the window (it shows an arrow hovering over a button) and then clicking on the button, icon or menu item you want to inspect in Firefox. When you return to the DOM Inspector you will see the node in the DOM tree that corresponds to the control you have just clicked. If this doesn't appear to work, check that you are inspecting the window you are clicking in – use File, Inspect Window.

From here, you can investigate the DOM tree and find out everything you need to know about that particular part of the user interface. For example, if you inspect the status bar in the Firefox window – the one that usually says 'Done' – you will see it flash with a red border. When you move back to the DOM inspector, you will see that this is a statusbarpanel with ID statusbar-display, which is a child node of a statusbar object with id status-bar. You can also find out that this has a child node xul:label with value 'Done'.

Armed with this knowledge, we can now add our very own statusbarpanel to the status bar. Open the XUL file and change it to read:

```
<?xml version="1.0"
encoding="UTF-8"?>
```

```
<!DOCTYPE overlay SYSTEM
"chrome://helloworld/locale/
helloworld.dtd">
<overlay id="helloworld-overlay"
xmlns="http://www.mozilla.org/
keymaster/
gatekeeper/there.is.only.xul">
<script src="overlay.js"/>
<menupopup id="menu_ToolsPopup">
<menuitem id="helloworld-hello"
label="Helloworld"
oncommand=
"Helloworld.
onMenuItemCommand();"/>
</menupopup>
<statusbar id="status-bar">
<statusbarpanel
id="helloworld-
statusHelloworld"
label="Hello World">
</statusbarpanel>
</statusbar>
</overlay>
```

The last four lines define an overlay for the existing statusbar, which has id "status-bar", with our own statusbarpanel, which has id "helloworld-statusHelloworld" and a label to display. If you want to see more details about the statusbarpanel control, look it up at www.xulplanet.com.

If you now save the XUL file, close all Firefox windows and reopen the browser you will see 'Hello World' in the status bar to the far right.

You can change the text displayed in your status bar using JavaScript in the usual way to interact with the DOM. For example, you could change the onMenuItemCommand method in the JS file to read:

```
onMenuItemCommand: function() {
alert("Hello World"); }
```

```
var myStatusbar=document.
getElementById(
"Helloworld-statusHelloworld");
myStatusbar.label=
"A dynamic Hello World";
},
```

Now when you reload the extension you will see an alert box and, after you have dismissed it, the status line message changes.

This example is typical of how you work with controls: find your control in the DOM using getElementById or a similar function, then use its properties to change it.

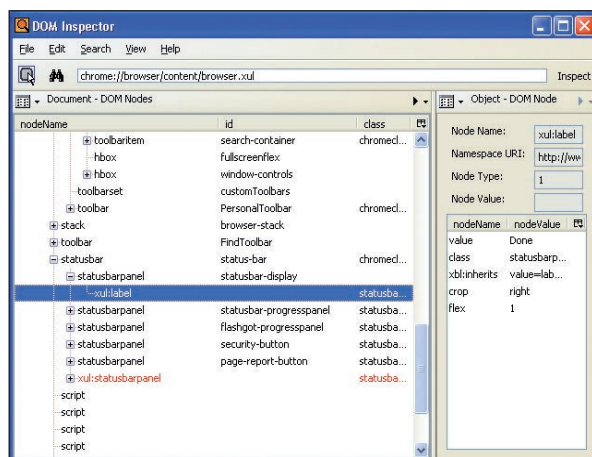
By now you should be familiar with the idea that, in this case, the DOM includes not only the webpage being displayed but the entire browser's user interface. The only thing that might puzzle you is how to get at the nodes of the webpage being displayed because, as the last example demonstrated, document.getElementById doesn't work directly with the DOM of the loaded webpage. The solution is that you have to use Content.document to get at the currently loaded HTML.

The DOM inspector is your best way to discover the names of user-interface components and to investigate its overall structure. Your second best friend is Venkman – the Mozilla JavaScript debugger. This works as an extension within Firefox and Thunderbird. If you plan to create your own extensions, you need to download it and use it to test and investigate your JavaScript. All you have to do is to make sure you are debugging the file browser.xul and then scroll down the loaded scripts until you find the name of the .js file for your extension – overlay.js in this case. This enables you to select any method defined in the file, set breakpoints and generally inspect how it works. If you can't find your JavaScript file even though your extension is loaded, the file contains a syntax error.

XPCOM

If you are writing an extension for Firefox, you may never have to get involved with XPCOM because you can do a lot simply by interacting and modifying the user interface. However, with Thunderbird in particular there often comes a time when you want to work with the raw data. In this case you need to learn XPCOM. Essentially this provides you with access to the internal workings of Firefox and Thunderbird. The XPCOM components are identified by their contract IDs, each of which starts with @mozilla.org in this case. To create a component you use the Components object, as in:

```
Var MyXPCOM=Components.Classes[
```



◀ The DOM Inspector shows the structure of the status panel

APRESS

Web Development Solutions Using Ajax, APIs, Libraries, and Hosted Services

★★★★

£23

From www.amazon.co.uk

Christian Heilmann and Mark Francis have written a very enjoyable book that keeps the reader engaged and excited about developing web projects. The problem is that they talk about creating novel, innovative websites as if it were easy and anyone could do it, skipping over technical details.

The book repeatedly points out that to do a good job you'll need to learn how to program, but without telling you how to do this, it feels as if it's missing some key information. The book glosses over topics, so while there's talk of using XML and Ajax, there's no information on the difficulties you're likely to encounter if you use any of these technologies. More technical information would have made this an invaluable reference tool.

As it stands, *Web Development Solutions* is still a cracking read. It's presented in such an exciting, tempting and positively motivating way that after reading only the first couple of chapters, you'll want to run straight out and build a stunning website for fame and profit. Building up this kind of interest is really a great achievement.

Of course, when you actually decide to start, you'll need to buy other books and perhaps even take courses on how to use the necessary technology. Still, if you need the motivation to get started or you want a glimpse of what's possible, this is a great read.

SUMMARY

- ✓ Creates enthusiasm for projects
- ✗ Glosses over difficulties

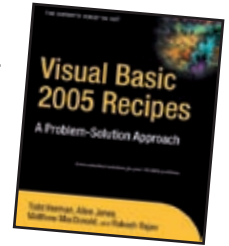
ISBN 1590598067
PAGES 280

APRESS

Visual Basic 2005 Recipes: A Problem-Solution Approach

★★★

£24

From www.amazon.co.uk

Visual Basic 2005 Recipes is designed to help programmers deal with common problems when programming in this language. This is a great idea for a reference book, but it is somewhat ruined by Todd Herman's pedestrian approach.

For starters, he never tackles any problems that are off the well-beaten track, and most of his solutions are obvious and too long. Neater solutions that teach you more about the language would have been more useful.

The presentation doesn't help. Recipe books are successful if they're easy to follow, but *Visual Basic 2005 Recipes* isn't. The book presents a problem and its solution, including some information on how it works. But the solution is usually just a program listing that you have to read through to understand what's happening.

The information on how the solution works is good to have, but to understand the text you must still read through the program listing. This can make it very difficult to understand, and learning from it is even harder. We'd have preferred a solution that illustrates what's wrong and explains how to fix it in greater detail.

The book covers a fairly wide range of topics and includes advanced subjects such as interop, working with Windows, encryption and the next version of Visual Basic. It may sound as if the book covers a lot, but it barely scratches the surface of each subject. It's OK as a basic reference book, but if you need something that delves deeper and explains better, look elsewhere.

SUMMARY

- ✓ Covers a wide range of topics
- ✗ A rather superficial approach

ISBN 1590598520
PAGES 664

```
"@mozilla.org/componentname"].  
createInstance();
```

If the component implements a standard service, you shouldn't create a new instance but use the existing one:

```
Var MyXPCOM=Components.Classes[  
"@mozilla.org/componentname"].  
getService();
```

Once you have a reference to an XPCOM object you can ask it to return a reference to any of the interfaces it supports using:

```
MyXPCOM.QueryInterface(  
Components.interfaces.  
requiredinterface);
```

As an example of using XPCOM, let's look at the prompt service. Although we used an alert window to pop up the 'hello world' message, this isn't the correct way to do the job. Extensions shouldn't use alerts because they are non-privileged HTML content code, and using them opens a potential security loophole. Instead, we should use the XPCOM prompt service. First, we get a reference to the currently running prompt service:

```
var PromptService = Components.  
classes[  
"@mozilla.org/embedcomp/prompt-  
service;1"]  
.getService();
```

Next we ask the prompt service to return a

reference to the nsIPromptService interface:

```
PromptService.QueryInterface(  
Components.interfaces.  
nsIPromptService);
```

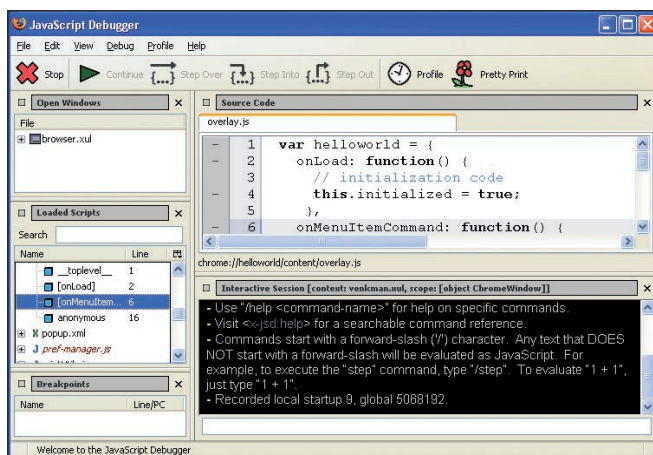
Now PromptService is a reference to a promptService interface or object, and we can use the alert method to pop up a window containing the appropriate message:

```
PromptService.alert(null,   
"Hello Extension","Hello World");
```

If you add this to the end of the onMenuItemCommand method in the JS file, you will see an additional alert-style dialog box pop up when you select the menu item.

Using other XPCOM components follows the same pattern: create or obtain a reference to the object, use the QueryInterface method to obtain the Interface you want and use the methods it provides. There are XPCOM components that enable you to access the email database, make a sound, work with files and access preferences, passwords and so on. To find out more about these, look them up in the reference section at www.xulplanet.com.

From here you should be able to develop your own extensions that do whatever you want. All it takes is a little patience. Search the web to find out what information is available, and then use the DOM Inspector and the JavaScript debugger to learn anything that isn't easily discovered in the documentation, or is completely undocumented.



◀ The JavaScript debugger checks HelloWorld for errors